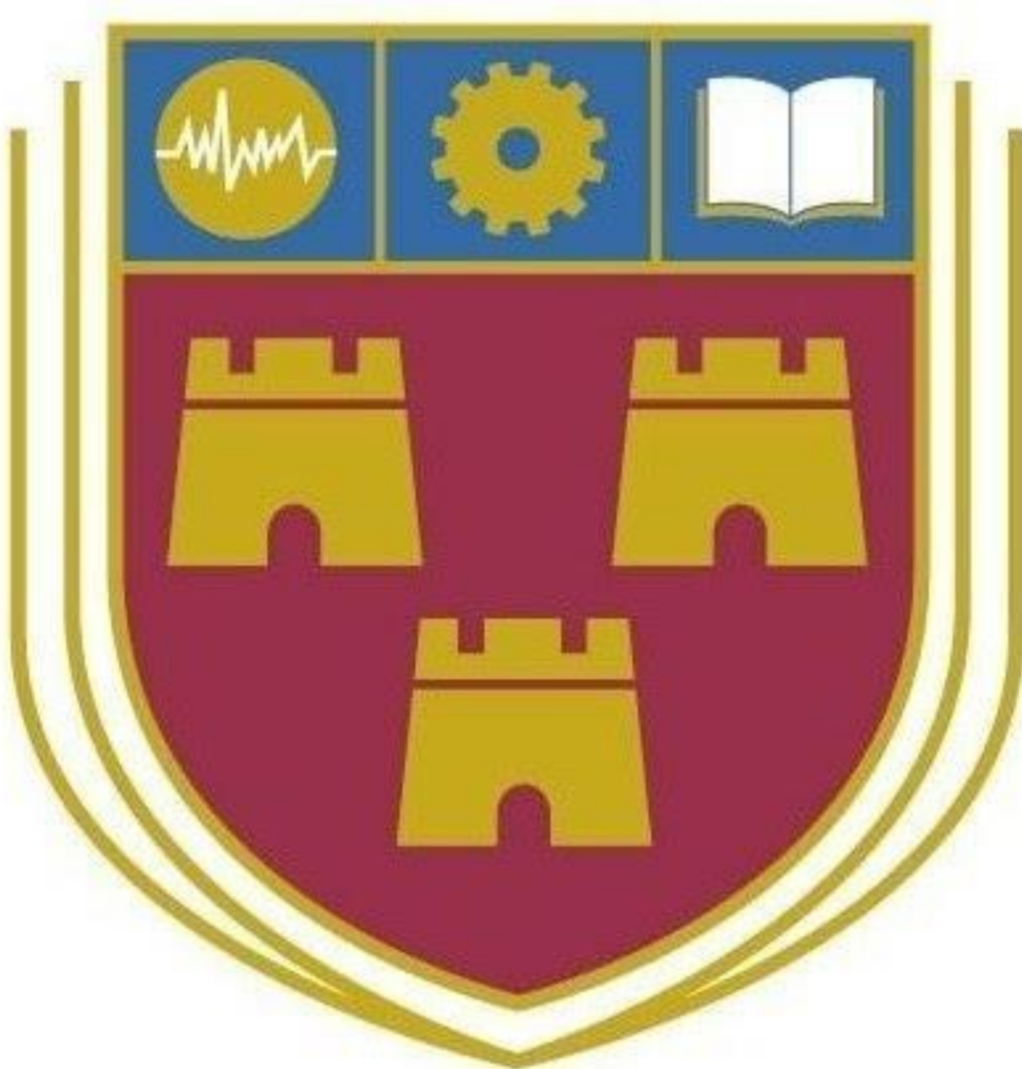
Step File Converter Technical Document

Alan Halpin - Institute of Technology Carlow

Started 25/04/2021



Supervisor: Joseph Kehoe

Student Number: C00229361

Abstract

The purpose of this document is to display all the code from this project. This tool is only a proof of concept that was unable to reach its full potential due to licensing issues with the SolidWorks API. This tool is still a work in progress.

The code displayed in this document was developed in Visual Studio 2019 and that is the format it has been displayed in.

Table of Contents

Abstract 2

Introducion 3

Project Code 4

- STEPFILE-Project.cpp 4
- STEP.h 6
- STEP.cpp 7
- FeatureFinder.h 11
- FeatureFinder.cpp 12

Plagiarism Declaration 20

- Declaration 20

Introducion

The STEP File Converter is a tool for CNC engineers, designed to automate a lengthy and costly process. Nothing like this application currently exists. The value is derived from the process being automated. This process involves performing vibration analysis on different permutations of an object until a satisfactory sequence of faces is found. Vibration analysis is performed to ensure the piece worked on remains stable throughout the cutting process. The proposed tool will extract geometrical data from a STEP file and catalogue the different faces of the object described. High level features will be identified from these faces. One by one these features will be subtracted from a solid block shape. Vibration analysis is performed between each step, if the piece does not pass the requirements then a different face will be used. This will eventually create a list of faces to cut in order. Ex. A,B,F,G,C might be an order in which to carve first. Although vibration analysis already exists within various CAD applications, the process described only exists in a manual implementation. This tool will use the Solid Works API to outsource the calculations and identify the optimal cutting order within the application. An engineer will pass a STEP file to the program and the program will start working on the results. Depending on the parts complexity this process may take some time. Typically this process is done manually and can take up to three weeks to complete, this tool hopes to automate that process and reduce that time down to one day.

Project Code

GitHub

STEPFILE-Project.cpp

```
/*
This work is licensed under the  Creative Commons Attribution-ShareAlike 4.0 International Licence.
To view of this licence, visit http://creativecommons.org/licenses/by-sa/4.0/.
*/
/*! \class STEPFILE-Project.cpp
    \brief System Controller
    \author Alan Halpin
    \date 29/04/2021
    \copyright Creative Commons Attribution-ShareAlike 4.0 International Licence
*/
// STEPFILE-Project.cpp : This file contains the 'main' function. Program execution begins and ends
there.
// SYSTEM CONTROLLER
#include <iostream>
#include <chrono>
#include <fstream>
#include <set>
#include <vector>
#include <algorithm>
#include <type_traits>
#include <filesystem>
#include "FeatureFinder.h"
#include "STEP.h"

using namespace std;
namespace fs = filesystem;

void printFeatureCoords(FeatureFinder featureObj, STEP stepDataObj)
{
    cout << "\n\nThe following high level features have been identified. Cuts need to be made at the
vertex points displayed below." << "\n\n";
    set<string> holder;
    for (auto hLFeature : featureObj.highLevelFeatures)
    {
        cout << "High Level Feature : " << hLFeature.first << "\n";
        for (auto face : hLFeature.second)
        {
            for (auto point : stepDataObj.vertexPoints[face])
            {
                holder.insert(point);
            }
        }
        for (auto item : holder)
        {
            cout << item << "\n";
        }
        holder.clear();
        system("pause");
    }
}

int main()
{
    string inputFile;
    STEP stepDataObj;
    // Read in STEP file names and display for user to choose from

    cout << "STEP File Location: STEPFILE-Project\\STEPFILES\\\n";
    string path = "STEPFILES/";

    for (const auto& entry : fs::directory_iterator(path))
        cout << entry.path() << endl;

    cout << "Enter the name of the STEP file (sans file extension)" << endl;
    cin >> inputFile;

    auto start = chrono::steady_clock::now();// Start Clock
```

```
    stepDataObj.stepController(inputFile);
    // Create featurefinder object and call the controller
    FeatureFinder highLevelFeatureObj;
    highLevelFeatureObj.featureFinderController(stepDataObj);

    auto end = chrono::steady_clock::now();// End Clock
    cout << "\nTime taken to read STEP: " << chrono::duration_cast<chrono::milliseconds>(end -
start).count() << "ms\n\n";

    system("pause");
    printFeatureCoords(highLevelFeatureObj, stepDataObj);
    return 0;
}
```

STEP.h

```
/*
This work is licensed under the  Creative Commons Attribution-ShareAlike 4.0 International Licence.
To view of this licence, visit http://creativecommons.org/licenses/by-sa/4.0/.
*/
/*! \class STEP.h
	\brief
	\author Alan Halpin
	\date 29/04/2021
	\copyright Creative Commons Attribution-ShareAlike 4.0 International Licence
*/
#pragma once
#include <map>
#include <set>
#include <string>
#include <vector>
using namespace std;
class STEP
{
private:
	map<string, string> stepDataList;
	map<string, vector<string>> edgeCurves;

	void extractFeatures(string inputFile);
	void checkDifference();
	void checkFacesThatTouch();
	//void findEdgeCurves();

public:
	vector<string> headerLines;
	set<string> diffLines;
	map<string, vector<string>> stepFeatureList;
	map<string, vector<string>> vertexPoints;
	map<string, vector<string>> cartesianPoints;
	map<string, vector<string>> touchingFaces;
	map<string, map<string, vector<string>>> edgeCurveGeometry;

	void stepController(string inputFile);
};
```

STEP.cpp

```
/*
This work is licensed under the  Creative Commons Attribution-ShareAlike 4.0 International Licence.
To view of this licence, visit http://creativecommons.org/licenses/by-sa/4.0/.
*/
/*! \class STEP.cpp
	\brief Create DataList and Identify low level features
	\author Alan Halpin
	\date 29/04/2021
	\copyright Creative Commons Attribution-ShareAlike 4.0 International Licence
*/
#include "STEP.h"
#include <iostream>
#include <fstream>
#include <map>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;
// This method is used to identify the lines in the step file that are unrelated to the adv face
features
// These lines are essential for building the STEP file
void STEP::checkDifference()
{
	map<string, vector<string>> featureList = STEP::stepFeatureList;
	map<string, string> dataList = STEP::stepDataList;
	set<string> featureLines;
	set<string> dataLines;
	set<string> result;

	for (auto key : featureList)
	{
		for (auto it = key.second.begin(); it != key.second.end(); ++it)
		{
			featureLines.insert(*it);
		}
	}
	for (auto key2 : dataList)
	{
		dataLines.insert(key2.second);
	}
	// Compare features to data to see what's missing
	set_difference(dataLines.begin(), dataLines.end(), featureLines.begin(),
featureLines.end(),inserter(result, result.begin()));
	STEP::diffLines = result;
}
// This method removes duplicates from a map of vectors
map<string,vector<string>> removeDuplicates(map<string, vector<string>> mapWithDupes)
{
	vector<string>::iterator rd;
	vector<string> v;
	for (auto &item : mapWithDupes)
	{
		v = item.second;
		sort(v.begin(), v.end());
		rd = unique(v.begin(), v.end());
		v.resize(distance(v.begin(), rd));
		mapWithDupes[item.first] = v;
	}
	return mapWithDupes;
}
// This method reads a step file and creates a datalist and feature list
void STEP::extractFeatures(string inputFile)
{
	// Declare Variables and DS
	string currentLine;
	string stepNumber;
	string multipleLineSTEP;
	map<string, string> dataList;
	map<string, vector<string>> featureList;
	vector<string> faces;
	ifstream StepFile;
```

```
	bool header = true;

	// First read the STEP file and insert lines into datalist
	string FilePath = ("STEPFILES/" + inputFile + ".step");
	std::cout << "Extracting Features... \n";
	StepFile.open(FilePath.c_str()); // Read STEP File

	if (!StepFile)
	{
		std::cout << "Unable to open STEP File\n";
		exit(1);
	}
	else
	{
		// Read STEP file's contents into memory
		std::cout << "STEP File Opened\n";
		while (getline(StepFile, currentLine))
// Cycle through each line
		{
			if (currentLine[0] == '#')
			{
				header = false;
				if (currentLine.find(";") != string::npos)
// if the line ends
				{
					stepNumber = currentLine.substr(0, currentLine.find(" "));
					dataList.insert({ stepNumber, currentLine });
// insert data section into map
					if (currentLine.find(" ADVANCED_FACE ") != string::npos)
					{
						faces.push_back(currentLine);
// insert adv face locations into a set for use later
						featureList[stepNumber].push_back(currentLine);
 // insert adv faces into feature list
					}
				}
				else // line doesn't end, it is a multiple line step
				{
					stepNumber = currentLine.substr(0, currentLine.find(" "));
					multipleLineSTEP = currentLine;
				}
				continue;
			}
			else if (header == true) // add to header
			{
				STEP::headerLines.push_back(currentLine);
				continue;
			}
			else if (multipleLineSTEP != "")
// If the step has multiple lines, append additional lines onto string
			{
				if (currentLine.find(";") != string::npos)
// If it is the last line (end is denoted by ";")
				{
					multipleLineSTEP.append(currentLine);
					dataList.insert({ stepNumber, multipleLineSTEP });
					multipleLineSTEP = "";
				}
				else // Not end, so append
				{
					multipleLineSTEP.append(currentLine);
				}
				continue;
			}
			else // Current line is not data or header, so continue
			{
				continue;
			}
		}
		STEP::stepDataList = dataList;
		stepNumber = "";
		currentLine = "";
		StepFile.close();
		std::cout << "STEP File Closed\n";
```

```
                // Declare Variables
                map<string, vector<string>> vPoints;
                map<string, vector<string>> cartPoints;
                vector<string> foundlines;
                vector<string> nextlines;
                set<string> subFeatures;
                bool numberFound = false;
                bool lastNumberFound = false;
                bool vPoint = false;

                // Store features with sub features next
                for (auto item : faces) // Iterate through adv faces
                {
                        currentLine = item.substr(0, item.find(" "));
// CurrentLine used to identify the current adv face
                        //std::cout << "\nNEW FACE " << " " << item << "\n";
                        nextlines.insert(nextlines.begin(), item);
// Must insert the adv face at the start of the list

                        while (lastNumberFound == false)
// runs until last number is found
                        {
                                for (auto nLine : nextlines)
// cycles through set nextlines
                                {
                                        if (nLine.find(" EDGE_CURVE ") != string::npos)
                                        {
                                                STEP::edgeCurves[currentLine].push_back(nLine);
                                        }
                                        if (nLine.find(" VERTEX_POINT ") != string::npos)
                                        {
                                                vPoint = true;
                                        }
                                        else if (nLine.find(" CARTESIAN_POINT ") != string::npos)
                                        {
                                                cartPoints[currentLine].push_back(nLine);
                                        }

                                        string subnLine = nLine.substr(nLine.find(" "),
nLine.size());
// get sub string to avoid reading step's own ID
                                        for (char& ch : subnLine)
// Cycles through each character in subnLine
                                        {
                                                if (numberFound == true)
                                                {
                                                        if (isdigit(ch))
                                                        {
                                                                stepNumber += ch;
                                                        }
                                                        else
                                                        {

//std::cout << "\nFound : #" + stepNumber;
                                                                foundlines.push_back(dataList["#" +
stepNumber]);
// add found numbers to foundlines for next iteration

        featureList[currentLine].push_back(dataList["#" + stepNumber]);
// Add found step into currentline's list of steps
                                                                if (vPoint == true)
                                                                {

        vPoints[currentLine].push_back(dataList["#" + stepNumber]);
                                                                        vPoint = false;
                                                                }
                                                                numberFound = false;
                                                                stepNumber = "";
                                                        }
                                                }
                                                else if (ch == '#')
                                                {
                                                        numberFound = true;
```

```
								}
							}
							if (foundlines.size() == 0)
							{
								lastNumberFound = true;
							}
						}
						nextlines.clear();
						for (int i = 0; i < foundlines.size(); i++)
							nextlines.push_back(foundlines[i]);
						foundlines.clear();
					} // end while
					lastNumberFound = false;
				}
				// Remove Duplicates
				removeDuplicates(featureList);
				removeDuplicates(vPoints);
				removeDuplicates(cartPoints);
				STEP::stepFeatureList = featureList;
				STEP::vertexPoints = vPoints;
				STEP::cartesianPoints = cartPoints;
				std::cout << "\nAdvanced Faces Found: " << faces.size() << "\n";
				checkDifference();
		}
}
// This method compares features to create a list of features that touch at one point or more
void STEP::checkFacesThatTouch()
{ // 2 faces touch if they share a vertex point with the same geometrical location (X,Y,Z)
	map<string, vector<string>> touchingFaces;
	string subLine, subLine1;
	for (auto step : STEP::vertexPoints)
	{
		for (auto step2 : STEP::vertexPoints)
		{
			if (step != step2)
			{
				for (auto item : step.second)
				{
					subLine = item.substr(item.find(" "), item.size());
					for (auto item2 : step2.second)
					{
						subLine1 = item2.substr(item2.find(" "),
item2.size());
						if (subLine == subLine1)
						{

	touchingFaces[step.first].push_back(step2.first);
							goto next;
						}
					}
				}
			}
		next:
			continue;
		}
	}
	STEP::touchingFaces = touchingFaces;

}
// This method controls the step class
void STEP::stepController(string inputFile)
{
	cout << "File name: " << inputFile << "\n";
	extractFeatures(inputFile);
	checkFacesThatTouch();
	//findEdgeCurves();
}
```

FeatureFinder.h

```
/*
This work is licensed under the  Creative Commons Attribution-ShareAlike 4.0 International Licence.
To view of this licence, visit http://creativecommons.org/licenses/by-sa/4.0/.
*/
/*! \class FeatureFinder.h
	\brief
	\author Alan Halpin
	\date 29/04/2021
	\copyright Creative Commons Attribution-ShareAlike 4.0 International Licence
*/
#pragma once
#include "STEP.h"
class FeatureFinder
{
private:
	long double minX, maxX;
	long double minY, maxY;
	long double minZ, maxZ;
	string minXorg, maxXorg, minYorg, maxYorg, minZorg, maxZorg;

	void findMinMax(STEP stepDataObj);
	void createCubeToFit(STEP cubeObj, STEP stepDataObj);
	void identifyHighLevelFeatures(STEP stepDataObj, STEP cubeObj);

public:
	void featureFinderController(STEP stepDataObj);
	map<int, set<string>> highLevelFeatures;

};
```

FeatureFinder.cpp

```
/*
This work is licensed under the  Creative Commons Attribution-ShareAlike 4.0 International Licence.
To view of this licence, visit http://creativecommons.org/licenses/by-sa/4.0/.
*/
/*! \class FeatureFinder.cpp
    \brief Identify High Level Features
    \author Alan Halpin
    \date 29/04/2021
    \copyright Creative Commons Attribution-ShareAlike 4.0 International Licence
*/
// This class will be used to identify high level features
#include <iostream>
#include <iomanip>
#include <fstream>
#include <filesystem>
#include "FeatureFinder.h"
#include "STEP.h"

using namespace std;
namespace fs = filesystem;
// This method finds the Min / Max points in the STEP file, to create an object that fits the min /
max of the original
void FeatureFinder::findMinMax(STEP stepDataObj)
{                                       //          x                       y                    z
    // #796 = CARTESIAN_POINT ( 'NONE',  ( 28.20906519726944239, 20.00000000000000000,
13.79214587795902425 ) ) ;
    FeatureFinder::minX = 10000000;
    FeatureFinder::maxX = -10000000;
    FeatureFinder::minY = 10000000;
    FeatureFinder::maxY = -10000000;
    FeatureFinder::minZ = 10000000;
    FeatureFinder::maxZ = -10000000;
    FeatureFinder::maxXorg, minXorg, maxYorg, minYorg, maxZorg, minZorg;
    string currentLine;
    string number;
    long double placeHolder;
    bool numberFound = false;
    int count = 0; // When count = 0, X && count = 1, Y && count = 2, Z

    for (auto key : stepDataObj.vertexPoints) // cycles through each feature
    {
        //cout << "\n" << "Key: " << key.first << " Results\n\n";
        for (auto it = key.second.begin(); it != key.second.end(); ++it)
// Cycles through each sub feature
        {
                currentLine = *it;
                string subnLine = currentLine.substr(currentLine.find("="), currentLine.size()); //
get sub string
                for (char& ch : subnLine)
// Cycles through each character in subnLine
                {
                    if (numberFound == true)
                    {
                        if (ch == ',' || ch == ' ')
// These charcters signal the end of the current number
                        {
                            char* char_arr;
                            char* end;
                            char_arr = &number[0];
                            placeHolder = strtod(char_arr, &end); // string to double
                            switch (count)
                            {
                            case 0: // X
                                if (placeHolder < minX)
                                {
                                    minX = placeHolder;
                                    minXorg = number;
                                }
                                else if (placeHolder > maxX)
                                {
                                    maxX = placeHolder;
                                    maxXorg = number;
```

```
                            }
                            break;
                        case 1: // Y
                            if (placeHolder < minY)
                            {
                                minY = placeHolder;
                                minYorg = number;
                            }
                            else if (placeHolder > maxY)
                            {
                                maxY = placeHolder;
                                maxYorg = number;
                            }
                            break;
                        case 2: // Z
                            if (placeHolder < minZ)
                            {
                                minZ = placeHolder;
                                minZorg = number;
                            }
                            else if (placeHolder > maxZ)
                            {
                                maxZ = placeHolder;
                                maxZorg = number;
                            }
                            break;
                        }
                        count++;
                        number = "";
                        numberFound = false;
                    }
                    else
                    {
                        number += ch;
                    }
                }
                else if (ch == '-' || isdigit(ch))
                {
                    number += ch;
                    numberFound = true;
                }
            }
            count = 0;
        }
    }
}
void writeFile(STEP cubeObj) // writing the cubeobj
{
    vector<string> header = cubeObj.headerLines;
    set<string> compileLines = cubeObj.diffLines;
    set<string> featureLines;
    ofstream TestFile("WriteTests/cubextended.step");

    for (auto key : cubeObj.stepFeatureList)
    {
        for (auto it = key.second.begin(); it != key.second.end(); ++it)
        {
            featureLines.insert(*it);
        }
    }
    for (auto line1 : header)
    {
        TestFile << line1 << "\n";
    }
    for (auto line2 : featureLines)
    {
         TestFile << line2 << "\n";
    }
    for (auto line3 : compileLines)
    {
        TestFile << line3 << "\n";
    }
    TestFile << "ENDSEC;\nEND - ISO - 10303 - 21;";
    TestFile.close(); // file closed
```

```
}

void FeatureFinder::createCubeToFit(STEP cubeObj, STEP stepDataObj)
{
    FeatureFinder cubeFinder;
    cubeFinder.findMinMax(cubeObj); // Find min / max of cube

    map < string, vector<string>> features = cubeObj.stepFeatureList;
    string number = "";
    string maxX = to_string(FeatureFinder::maxX);
    string maxY = to_string(FeatureFinder::maxY);
    string maxZ = to_string(FeatureFinder::maxZ);
    string minX = to_string(FeatureFinder::minX);
    string minY = to_string(FeatureFinder::minY);
    string minZ = to_string(FeatureFinder::minZ);
    string stepNumber;
    string stepNumber1;
    string newValue;
    bool numberFound = false;
    long double placeHolder;
    int count = 0;
    char start;

    // #796 = CARTESIAN_POINT ( 'NONE',  ( 28.20906519726944239, 20.00000000000000000,
13.79214587795902425 ) ) ;

    for (auto &item : cubeObj.stepFeatureList) // cycles through each set of vectors
    {
        //std::cout << "\n\n" << item.first << "\n\n";
        for (auto &item2 : item.second) // cycles through each element of the vector
        {
            for (auto searchItem : cubeObj.cartesianPoints)
// cycles through each set of vectors
            {
                for (auto searchItem2 : searchItem.second)
// cycles through each element of the vector
                {
                    if (item2 == searchItem2) // If the current item === the search item. Then replace
XYZ
                    {
                        string subnLine = item2.substr(item2.find("="), item2.size());
                        //cout << "BEFORE: " << item2 << "\n";
                        for (char& ch : subnLine) // Cycles through each character in item2
                        {
                            if (numberFound == true)
                            {
                                if (ch == ',' || ch == ' ') // These charcters signal the end of the
current number
                                {
                                    char* char_arr;
                                    char* end;
                                    char_arr = &number[0];
                                    placeHolder = strtod(char_arr, &end); // string to double
                                    switch (count)
                                    {
                                    case 0: // X
                                        if (placeHolder == cubeFinder.minX)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::minXorg);
                                            item2 = newValue;
                                        }
                                        else if (placeHolder == cubeFinder.maxX)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::maxXorg);
                                            item2 = newValue;
                                        }
                                        break;
                                    case 1: // Y
                                        if (placeHolder == cubeFinder.minY)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::minYorg);
```

```
                                            item2 = newValue;
                                        }
                                        else if (placeHolder == cubeFinder.maxY)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::maxYorg);
                                            item2 = newValue;
                                        }
                                        break;
                                    case 2: // Z
                                        if (placeHolder == cubeFinder.minZ)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::minZorg);
                                            item2 = newValue;
                                        }
                                        else if (placeHolder == cubeFinder.maxZ)
                                        {
                                            newValue = item2.replace(item2.find(start,
item2.find(number)), number.size(), FeatureFinder::maxZorg);
                                            item2 = newValue;
                                        }
                                        break;
                                    }
                                    count++;
                                    number = "";
                                    numberFound = false;
                                }
                                else
                                {
                                    number += ch;
                                }

                            }
                            else if (ch == '-' || isdigit(ch))
                            {
                                numberFound = true;
                                number += ch;
                                start = ch;
                            }
                        }
                        count = 0;
                        //std::cout << "REPLACED: " << item2 << "\n";
                    }
                    else {
                        continue;
                    }
                }
            }
        }
    }
    identifyHighLevelFeatures(stepDataObj, cubeObj);
}

// This method checks if two points are equal within a range
bool isEqual(string pointOne , string pointTwo)
{
    string oneX, oneY, oneZ;
    string twoX, twoY, twoZ;
    string number;
    int count = 0;
    bool numberFound = false;
    for (auto ch : pointOne)
    {
        if (numberFound == true)
        {
            if (ch == ',' || ch == ' ')
            {
                switch (count)
                {
                case 0:
                    oneX = number;
                    break;
                case 1:
```

```
                    oneY = number;
                    break;
                case 2:
                    oneZ = number;
                    break;
                }
                count++;
                number = "";
                numberFound = false;
            }
            else
            {
                number += ch;
            }
        }
        else if (ch == '-' || isdigit(ch))
        {
            numberFound = true;
            number += ch;
        }
    }
    count = 0;
    for (auto ch : pointTwo)
    {
        if (numberFound == true)
        {
            if (ch == ',' || ch == ' ')
            {
                switch (count)
                {
                case 0:
                    twoX = number;
                    break;
                case 1:
                    twoY = number;
                    break;
                case 2:
                    twoZ = number;
                    break;
                }
                count++;
                number = "";
                numberFound = false;
            }
            else
            {
                number += ch;
            }
        }
        else if (ch == '-' || isdigit(ch))
        {
            numberFound = true;
            number += ch;
        }
    }
    if (oneX.substr(0, 15) == twoX.substr(0, 15))
    {
        if (oneY.substr(0, 15) == twoY.substr(0, 15))
        {
            if (oneZ.substr(0, 15) == twoZ.substr(0, 15))
            {
                return true;
            }
        }
    }
    return false;
}

// This method writes the HLF to their own step files.
void write(map<int, set<string>> highLevelFeatures, STEP stepDataObj)
{ // write the objects
    // remove files already inside first
    string path = "WriteTests/";
    for (const auto& entry : fs::directory_iterator(path))
```

```
            fs::remove_all(entry.path());
    vector<string> header = stepDataObj.headerLines;
    set<string> compileLines = stepDataObj.diffLines;
    set<string> featureLines;
    for (auto key : highLevelFeatures)
    {
        string name = to_string(key.first);
        string FilePath = ("WriteTests/Object" + name + ".step");
        ofstream TestFile(FilePath.c_str());
        for (auto item : key.second)
        {
            for (auto item2 : stepDataObj.stepFeatureList[item])
            {
                featureLines.insert(item2);
            }
        }
        for (auto line1 : header)
        {
            TestFile << line1 << "\n";
        }
        for (auto line2 : featureLines)
        {
            TestFile << line2 << "\n";
        }
        for (auto line3 : compileLines)
        {
            TestFile << line3 << "\n";
        }
        TestFile << "ENDSEC;\nEND - ISO - 10303 - 21;";
        TestFile.close(); // file closed
        featureLines.clear();
    }
    cout << "All Features have been written to STEPFILE-Project/WriteTests" << endl;
}

void FeatureFinder::identifyHighLevelFeatures(STEP stepDataObj, STEP cubeObj)
{
    // First, compare with the cube created to see what points are not shared.
    set<string> linesNotShared;
    string subLine, subLine1, subSearch, subSearch1;
    bool found = false;

    for (auto advFace : stepDataObj.stepFeatureList)
    {
        for (auto vPoint : advFace.second)
        {
            for (auto searchItem : stepDataObj.vertexPoints[advFace.first])
            {
                subSearch = searchItem.substr(0, searchItem.find(" "));
                if (subSearch == vPoint.substr(0, vPoint.find(" ")))
                {
                    subLine = vPoint.substr(vPoint.find("("), vPoint.size());
                    //cout << "\n\n" << vPoint << "\n\n\n";
                    for (auto advFace1 : cubeObj.stepFeatureList)
                    {
                        for (auto vPoint2 : advFace1.second)
                        {
                            for (auto searchItem2 : cubeObj.vertexPoints[advFace1.first])
                            {
                                subSearch1 = searchItem2.substr(0, searchItem2.find(" "));
                                if (subSearch1 == vPoint2.substr(0, vPoint2.find(" ")))
                                {
                                    subLine1 = vPoint2.substr(vPoint2.find("("), vPoint2.size());
                                    if (isEqual(subLine, subLine1))
                                    {
                                        found = true;
                                    }
                                    else
                                    {
                                        continue;
                                    }
                                }
                            }
```

```
                    }
                }
                if (found == false)
// Current point has not been found, add to list
                {
                    linesNotShared.insert(vPoint);
                }
                else {
                    found = false;
                }
            }
            else {
                continue;
            }
        }
    }
}

// Which faces do these points belong to?
map<string, set<string>> points;
for (auto advFace : stepDataObj.stepFeatureList)
{
    for (auto item : linesNotShared)
    {
        for (auto step : advFace.second)
        {
            if (item == step)
            {
                points[advFace.first].insert(item);
            }
        }
    }
}
int pointCount = 0;
set <string> features;
// check if all of faces vPoints can be found in Points not shared
for (auto advFace : points)
{
    for (auto vPoint : stepDataObj.vertexPoints[advFace.first])
    {
        subLine = vPoint.substr(0, vPoint.find(" "));
        for (auto vertex : linesNotShared)
        {
            subLine1 = vertex.substr(0, vertex.find(" "));
            if (subLine == subLine1)
            {
                pointCount++;
            }
            if (pointCount == stepDataObj.vertexPoints[advFace.first].size())
            {
                features.insert(advFace.first);
                //cout << advFace.first << "\n";
                goto NextPoint;
            }
        }
    }
NextPoint:
    pointCount = 0;
    continue;
}

// check if advFaces found have faces that touch in common
set<string> features2 = features;
set<string> toremove;
map<int, set<string>> highLevelFeatures;
int objectNo = 0;

for (auto face : features)
{
    if (find(toremove.begin(), toremove.end(), face) != toremove.end())
// if the current face has already been inserted
    {
        continue;
    }
```

```
        else // if it hasn't, insert it now
        {
            highLevelFeatures[objectNo].insert(face);
            toremove.insert(face);
        }
        if (features2.empty()) // if there are no more features to search from
        {
            continue;
        }
        for (auto face2 : features2)
        {
            if (face != face2) // don't compare the same face
            {
                // check if face is touching face2
                if (find(stepDataObj.touchingFaces[face].begin(),
stepDataObj.touchingFaces[face].end(), face2) != stepDataObj.touchingFaces[face].end())
                {
                    highLevelFeatures[objectNo].insert(face2);
                    toremove.insert(face2);
                }
            // check if face and face2 have a common face (that is listed in features)
                else
                {
                    for (auto common : stepDataObj.touchingFaces[face])
// check which faces touch face
                    {
                        for (auto common2 : stepDataObj.touchingFaces[face2])
                        {
                            if (common == common2 && find(features.begin(), features.end(), common) !=
features.end())
                            {
                                highLevelFeatures[objectNo].insert(face2);
                                toremove.insert(face2);
                            }
                        }
                    }
                }
            }
        }
        for (auto item : toremove)
        {
            features2.erase(item);
        }
        //toremove.clear();
        objectNo++;
    }
    FeatureFinder::highLevelFeatures = highLevelFeatures;
    if (highLevelFeatures.empty())
    {
        cout << "No Features found\n";
        return;
    }
    else
    {
        write(highLevelFeatures, stepDataObj);
    }
}

void FeatureFinder::featureFinderController(STEP stepDataObj)
{
    cout << "Welcome to the Feature Finder\n";
    findMinMax(stepDataObj);
    STEP cubeObj; // Create Cube Objects
    cubeObj.stepController("Cube"); // Read cube and fill variables
    createCubeToFit(cubeObj, stepDataObj);
    //writeFile(cubeObj);
}
```

Plagiarism Declaration

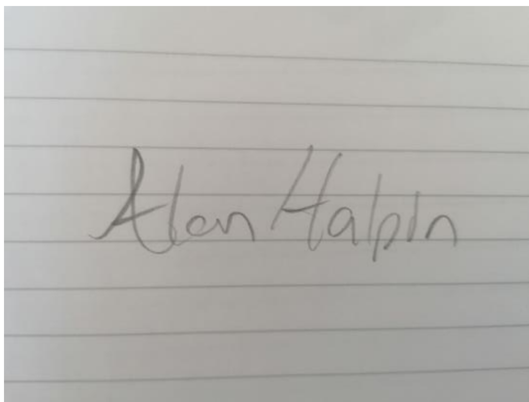
Declaration

- I declare that all material in this submission e.g.thesis/essay/project/assignment is entirely my/our own work except where duly acknowledged.
- I have cited the sources of all quotations, paraphrases, summaries of information, tables, diagrams or other material; including software and other electronic media in which intellectual property rights may reside.
- I have provided a complete bibliography of all works and sources used in the preparation of this submission.
- I understand that failure to comply with the Institute's regulations governing plagiarism constitutes a serious offense.

Student Name: Alan Halpin

Student Number: C00229361

Signature:



Date: 30/04/2021